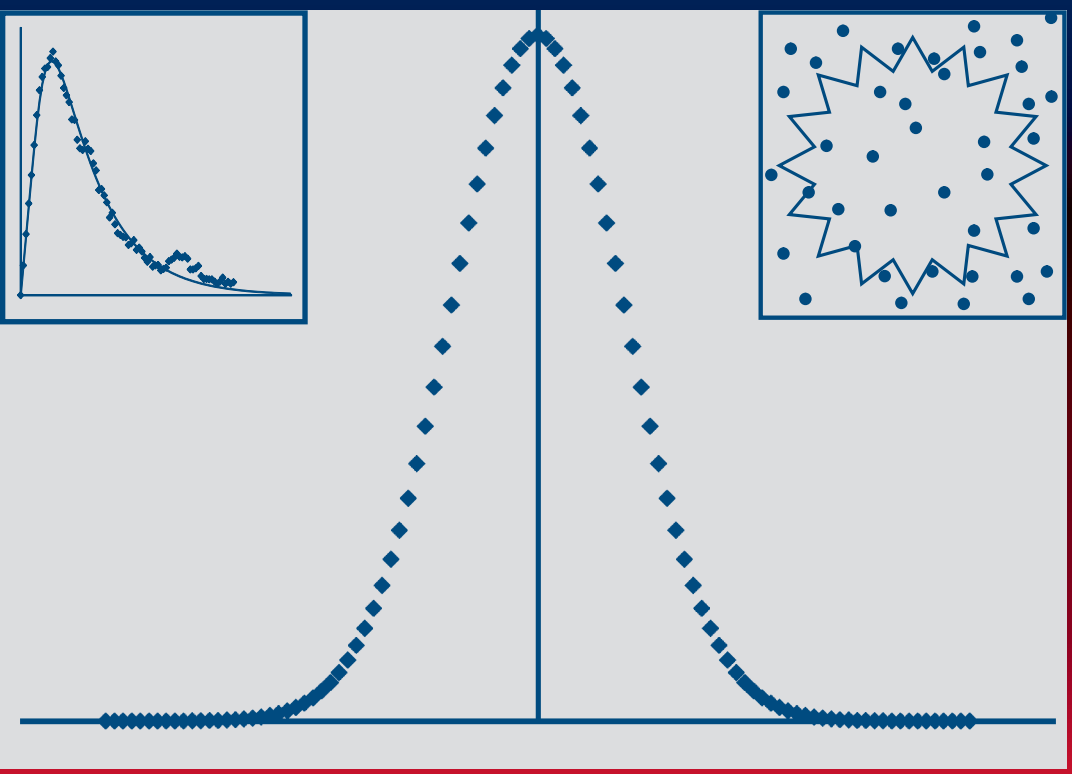
Уральский федеральный университет
имени первого Президента России Б.Н.Ельцина
Физико-технологический институт

К. А. НЕКРАСОВ
С. И. ПОТАШНИКОВ
А. С. БОЯРЧЕНКОВ
А. Я. КУПРЯЖКИН

# МЕТОД МОНТЕ-КАРЛО НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

Учебное пособие



Министерство образования и науки Российской Федерации
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

К. А. Некрасов, С. И. Поташников,
А. С. Боярченков, А. Я. Купряжкин

# Метод Монте-Карло на графических процессорах

Учебное пособие

Рекомендовано методическим советом УрФУ
для студентов, обучающихся по направлениям подготовки
14.04.02 — Ядерная физика и технологии;
09.04.02 — Информационные системы и технологии;
14.04.01 — Ядерные реакторы и материалы

Екатеринбург
Издательство Уральского университета
2016

УДК 519.245:004(075.8)
ББК 22я73+32.97я73
М54

Авторы:
Некрасов К. А., Поташников С. И., Боярченков А. С., Купряжкин А. Я.

Рецензенты:
Институт теплофизики УрО РАН (д-р физ.-мат. наук, проф. *В. Г. Байдаков*); гл. науч. сотр. лаборатории математического моделирования Института промышленной экологии УрО РАН д-р физ.-мат. наук, проф. *А. Н. Вадаксин*

**Метод Монте-Карло на графических процессорах** : учебное посо-
М54 бие / К. А. Некрасов, С. И. Поташников, А. С. Боярченков, А. Я. Купряжкин. — Екатеринбург : Изд-во Урал. ун-та, 2016. — 60 с.

ISBN 978-5-7996-1723-3

В учебном пособии изложены основные идеи метода Монте-Карло, принципы распараллеливания расчетов, организации высокоскоростных параллельных вычислений методом Монте-Карло на графических процессорах NVIDIA архитектуры CUDA. Подробно проанализирован пример моделирования методом Монте-Карло диффузии нейтронов через пластину.

Пособие предназначено для проведения практических занятий по программированию графических процессоров для магистрантов по направлениям подготовки 09.04.02, 14.04.02, специалистов по направлению подготовки 141401.

Библиогр.: 15 назв. Рис. 15. Прил. 1.

УДК 519.245:004(075.8)
ББК 22я73+32.97я73

ISBN 978-5-7996-1723-3

# Введение

Одним из широко используемых подходов к численному моделированию физических систем является метод Монте-Карло [1]–[2], для применения которого необходим анализ большого количества случайных состояний и вариантов поведения исследуемой системы, что связано с большими объемами вычислений.

Варианты случайного поведения системы, рассматриваемые методом Монте-Карло, обычно генерируются и обрабатываются по одному и тому же общему алгоритму, будучи при этом независимыми друг от друга. Это позволяет эффективно реализовывать метод Монте-Карло на поточно-параллельных вычислительных системах.

В последние годы появилась практическая возможность проведения поточно-параллельного физико-математического моделирования на графических процессорах (GPU) — вычислительных устройствах параллельной архитектуры, изначально предназначенных для отображения графики на персональных компьютерах и рабочих станциях [3].

В настоящее время ведущие производители графических процессоров сами позиционируют новые GPU как универсальные системы, предназначенные не только для графических вычислений, но и для расчетов общего назначения. В частности, компанией NVIDIA выпускаются графические процессоры архитектуры CUDA [4] и соответствующее программное обеспечение [5], которые позволяют сравнительно легко программировать GPU на языке, являющемся расширением языка C, сохраняющем синтаксис последнего и включающем все его основные возможности.

Современные графические процессоры имеют в своем составе до нескольких сотен параллельных процессоров, каждый из которых по вычислительной производительности сравним с центральным процессором персонального компьютера [6]– [7].

Опыт показывает, что при решении хорошо распараллеливаемых задач реальная производительность GPU оказывается не менее чем на два порядка выше, чем у современных центральных процессоров ПК. Отметим, что затраты на приобретение и эксплуатацию одного вычислительного комплекса с GPU примерно совпадают с затратами на обычный персональный компьютер, тогда как кластер эквивалентной производительности занимал бы намного больше места, будучи при этом намного сложнее в эксплуатации и на порядки дороже.

В работах [8]–[10] представлены результаты применения графических процессоров для моделирования кристаллов диоксида урана методом молекулярной динамики. Использование графических процессоров позволило ускорить решение данной задачи (по сравнению с центральным процессором) более чем в 600 раз. Настоящее пособие посвящено оценке возможностей графических процессоров архитектуры CUDA (*Compute Unified Device Architecture* [3]–[5]) на примере метода Монте-Карло, в частности при моделировании одномерной диффузии нейтронов через пластину.

Архитектуры графических процессоров, выпускаемых ведущими производителями — компаниями NVIDIA и AMD/ATI, несколько различны, что особенно проявляется при неграфических вычислениях. Наиболее удобной для таких вычислений мы считаем архитектуру CUDA [3]–[5], предлагаемую компанией NVIDIA.

В настоящем пособии будут рассмотрены особенности параллельной реализации метода Монте-Карло для высокопроизводительных вычислений на графических процессорах. При этом мы ограничимся программированием для CUDA, так как в настоящее время эта платформа представляется наиболее эффективной, а реализованный в ней подход — наиболее перспективным.

# 1. Применение метода Монте-Карло для моделирования физических систем

## 1.1. Содержание метода Монте-Карло

### 1.1.1. Метод Монте-Карло на примере вычисления площадей

Простейшей иллюстрацией метода Монте-Карло [1] является вычисление площади плоской фигуры ($S$) по принципу, показанному на рис. 1.1 [2].

Фигура заключена в квадрат, по которому случайным образом разбрасывается большое количество точек. Если число точек достаточно велико, то отношение площади $S$ к площади квадрата $L^2$ будет с заданной точностью равняться отношению количества точек $N'$, попавших внутрь фигуры, к полному количеству точек $N$

$$S \approx L^2 \frac{N'}{N}.$$

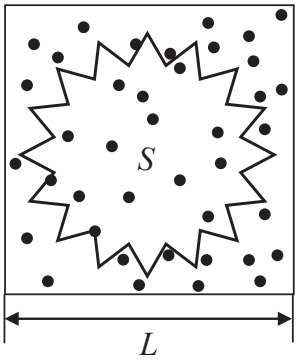



Рис. 1.1. Вычисление площади плоской фигуры с криволинейной границей методом Монте-Карло [2]

Точность оценки площади возрастает вместе с количеством точек $N$. Погрешность вычислений обычно пропорциональна $N^{-1/2}$.

Площадь фигуры можно интерпретировать как ее объем в двухмерном пространстве. Очевидно, что метод, показанный на рис. 1.1, может быть применен и для вычисления объема трехмерной фигуры, заключенной внутрь куба. Аналогично можно рассчитывать объемы многомерных фигур в пространствах бóльшей размерности. Этот подход может быть без модификации применен и для многомерного численного интегрирования, поскольку определенные интегралы могут быть представлены как объемы многомерных тел.

### 1.1.2. Распределения случайных величин

Одним из основных этапов любой вариации метода Монте-Карло является генерирование случайных величин, распределенных по необходимому закону. При вычислении площади фигуры в параграфе 1.1.1 этими случайными величинами были координаты точек, разбрасываемых по площади квадрата. Каждой точке соответствовали две координаты ($x$, $y$), каждая из которых должна была быть случайной величиной, равномерно распределенной на интервале (0; $L$).

Пусть некоторая случайная величина $\xi$ определена на интервале ($a$; $b$). *Плотностью распределения* $\xi$ называют функцию $w(x)$, такую, что

$$W(x) = \int_a^x w(x)dx = P(\xi < x) \ —$$

вероятность того, что $\xi$ окажется меньше, чем $x$. Сам интеграл $W(x)$ называется *интегральной (кумулятивной) функцией* распределения величины $\xi$, значения $W(x)$ всегда принадлежат интервалу (0; 1). Поскольку $P(\gamma \le b) = 1$, для любой плотности распределения выполняется условие нормировки

$$\int_a^b w(x)dx = 1. \qquad (1.1)$$

Несколько нестрого равномерно распределенными можно назвать те случайные величины, все допустимые значения которых равновероятны. Соответственно равномерно распределенные величины характеризуются плотностью распределения, не зависящей от $x$:

$$w(x) = \begin{cases} \dfrac{1}{b-a}, \text{ при } x \in (a; b); \\ 0, \quad \text{при } x \notin (a; b). \end{cases}$$

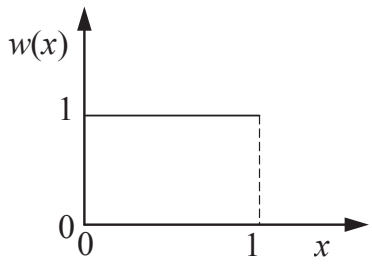



Рис. 1.2. Плотность вероятности для равномерного распределения на интервале (0; 1)

Такое распределение имеет прямоугольную форму, как это показано для интервала (0; 1) на рис. 1.2.

Кроме равномерно распределенных, широко используются и часто рассматриваются нормально распределенные (распределенные по Гауссу) случайные ве-

личины. Нормальной (гауссовой) случайной величиной называется случайная величина $\xi$, определенная на всей числовой оси $(-\infty;\infty)$ с плотностью распределения

$$w(x)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left[-\frac{\left(x-\overline{\xi}\right)^2}{2\sigma^2}\right],$$

где $\sigma$ — дисперсия $\overline{\xi}$, характеризующая разброс значений относительно среднего; $\overline{\xi}$ — математическое ожидание (среднее значение) $\xi$.

Среднее значение произвольной непрерывной случайной величины $\xi$ с плотностью распределения $w(x)$ может быть вычислено по формуле

$$\overline{\xi}=\int_0^1 xw(x)dx,$$

а дисперсия определяется по выражению

$$\sigma^2[\xi]=\int_0^1 x^2w(x)dx-\left(\overline{\xi}\right)^2.$$

### 1.1.3. Получение последовательностей случайных чисел с требуемой функцией распределения

Стандартные генераторы случайных чисел (см. подглаву 1.2) обычно создают числовые последовательности с равномерным распределением. В то же время для решения конкретных задач методом Монте-Карло могут потребоваться случайные величины, распределенные по самым разным законам. Существуют достаточно простые способы перехода от равномерно распределенной случайной последовательности к последовательностям с любыми другими распределениями. Остановимся на одном из них.

Пусть необходимо генерировать значения случайной величины $\xi$, распределенной в интервале $(a; b)$ с плотностью вероятности $w(x)$ и функцией распределения $W(x)$. Значения функции распределения $W(x)$ обязательно приходятся на интервал (0; 1), что позволяет сопоставить ей случайную величину $\gamma$, равномерно распределенную на этом же интервале.

Известно, что если $\gamma$ — случайная величина, равномерно распределенная на интервале (0; 1), то случайная величина $\xi \in (a; b)$, удовлетворяющая соотношению

$$W(\xi) \equiv \int_a^{\xi} w(x)dx = \gamma, \qquad (1.2)$$

будет распределена именно с плотностью вероятности $w(x)$ и функцией распределения $W(x)$.

В результате аналитического либо численного решения уравнения можно на основании имеющейся равномерно распределенной последовательности $\{\gamma_i\}$ получить случайную последовательность $\{\xi_i\}$ с требуемой плотностью распределения $w(x)$.

Изложенный метод не является единственным. Существуют и другие методы получения как непрерывных, так и дискретных случайных величин с требуемыми распределениями. В любом случае для получения разнообразных случайных последовательностей, которые могут потребоваться для реализации конкретных вариантов метода Монте-Карло, необходимо иметь качественный генератор какой-либо стандартной случайной последовательности, например, генератор равномерно распределенных случайных чисел.

## 1.2. Генераторы случайных чисел

### 1.2.1. Требования, предъявляемые к генераторам случайных чисел

В задачах, решаемых методом Монте-Карло, очень большое значение может иметь качество используемых случайных последовательностей. Как отличие распределения случайной величины от требуемого, так и коррелированность (взаимозависимость) соседних значений искажают результаты моделирования, причем не всегда очевидным образом. Случайные последовательности, подходящие для метода Монте-Карло, должны удовлетворять следующим требованиям:

- максимально точное воспроизведение заданного случайного распределения (в частности, равномерного распределения);

- отсутствие корреляций во взаимном расположении членов последовательности;
- достаточно большой период повторения (зацикливания) последовательности (или отсутствие такого периода, как у «генераторов энтропии»);
- одинаковая случайность всех битов числа;
- необратимость последовательности.

### 1.2.2. Типы генераторов случайных чисел

По принципу формирования числовой последовательности принято выделять следующие типы генераторов случайных чисел:

- генераторы, основанные на естественных процессах, таких как шум транзисторов или сетевого напряжения (*генераторы энтропии*). Не имеют периода. Создают заведомо случайные, поэтому очень качественные последовательности. Вместе с тем такие генераторы бывают неудобны на практике из-за таких ограничений:
  - как сравнительно медленная работа;
  - сложность распараллеливания;
  - сложность самостоятельной реализации на новых вычислительных устройствах, таких как графические процессоры;
  - невоспроизводимость случайных последовательностей;
- генераторы квазислучайных чисел. На основе простых функций создают числовые последовательности, члены которых заведомо коррелируют между собой. Такие генераторы могут применяться для решения задач, в которых значение имеет только равномерность распределения случайных чисел, а их корреляции неважны. Например, последовательность {1, 2, 3, 4, 5, 5, 4, 3, 2, 1} имеет вполне равномерное распределение, хотя и неслучайна;
- генераторы псевдослучайных чисел. Наиболее широко используются при моделировании случайных последовательностей на ЭВМ. Для получения следующих членов последовательности обычно осуществляют более или менее сложные операции над предыдущими членами, формируя очередное случайное число как комбинацию из цифр мантиссы результата этих операций, либо как комбинацию битов, представляющих результат.

Поскольку последующие члены последовательности основаны на предыдущих, генераторы псевдослучайных чисел имеют конечный период, который, впрочем, может быть очень большим. Степень равномерности распределения и коррелированность последовательности определяются особенностями конкретных алгоритмов.

Лучшие из генераторов псевдослучайных чисел, согласно известным тестам, достаточно хороши для всех существующих приложений метода Монте-Карло. При этом они имеют перед генераторами энтропии такие преимущества:

- высокую скорость работы;
- возможность одновременного исполнения многих генераторов в параллельных вычислительных потоках, в том числе на графических процессорах.

### 1.2.3. Особенности применения генераторов случайных чисел для реализации метода Монте-Карло на графических процессорах

Как отмечено выше, при параллельной реализации метода Монте-Карло на графическом процессоре создается большое количество параллельных вычислительных потоков (исполняемых на параллельных «вычислителях»), каждый из которых обрабатывает один (или несколько) случайных вариантов поведения или состояния модельной системы. Поскольку эти варианты должны быть максимально независимы друг от друга, вычислительные потоки не могут использовать одну и ту же последовательность случайных чисел: каждому потоку нужен свой генератор.

В принципе последовательности псевдослучайных чисел для каждого из вычислительных потоков могли бы быть сгенерированы заранее, а при необходимости — загружаться в память, доступную графическому процессору. Такое решение не представляется эффективным, поскольку доступ графического процессора к видеопамяти, где должны будут храниться эти числа, осуществляется слишком медленно; количество доступных случайных чисел становится ограниченным.

Следовательно, генератор случайных чисел должен быть запрограммирован как часть вычислительного ядра, исполняемого графическим процессором в параллельном режиме. При этом внутри каждого вы-

числительного потока генерируемая последовательность случайных чисел должна создаваться уникальным образом во избежание корреляции с остальными последовательностями.

Архитектура CUDA позволяет реализовывать хорошие генераторы псевдослучайных чисел на GPU. Одновременно эта архитектура обеспечивает идентификацию конкретных вычислительных потоков в ходе исполнения программы, необходимую для уникальной инициализации каждого из параллельных генераторов.

Ниже мы будем использовать генератор псевдослучайных чисел *Mersenne Twister* [11], [12]. Такой генератор является одним из лучших согласно известным тестам [13]–[14], он очень хорошо удовлетворяет всем требованиям из параграфа 1.2.1. Очень важно то, что авторами Mersenne Twister была специально разработана методика создания множества экземпляров данного генератора, которые могли бы работать параллельно по одному и тому же алгоритму, но при этом генерировать практически некоррелирующие случайные последовательности [15] (см. подробности ниже).

### 1.2.4. Генератор псевдослучайных чисел Mersenne Twister

**Принцип получения псевдослучайных чисел.** В качестве наглядного примера рассмотрим сначала один из простейших генераторов псевдослучайных чисел — метод середины квадратов, предложенный Дж. Нейманом в 1946 г. Пусть $\gamma_0$ — число, инициализирующее случайную последовательность, $\gamma_0 = 0.9876$. Возведение его в квадрат дает восьмизначное число $\gamma_0^2 = 0.97535376$. Четыре средние цифры этого числа можно использовать как следующий элемент псевдослучайной последовательности $\gamma_1 = 0.5353$. Дальнейшая генерация аналогична описанному $\gamma_1^2 = 0.28654609 \Rightarrow \gamma_2 = 0.6546$, и так далее.

Этот пример демонстрирует основные особенности генераторов псевдослучайных чисел:

- использование предыдущих членов последовательности для получения следующих;
- получение очередного числа как части мантиссы некоторого промежуточного результата;
- существование конечного периода последовательности. В показанном примере последовательность повторится полностью

после появления числа, совпадающего с одним из предыдущих (так как $\{\gamma_{i+1} = f(\gamma_i), \gamma_{i+2} = f(\gamma_{i+1}), ...\}$, то совпадение $\gamma_k = \gamma_i$ приведет к тому, что $\gamma_{k+1} = f(\gamma_k) = f(\gamma_i)$, и так далее).

**Алгоритм генератора Mersenne Twister.** Принцип работы генератора псевдослучайных чисел Mersenne Twister [12], [15], который мы в дальнейшем будем использовать, похож на показанный выше. Этот алгоритм относится к числу генераторов, основанных на линейных рекуррентных соотношениях, связывающих последующие элементы случайной последовательности $\{x_i\}$ с предыдущими.

Элементы последовательности $x_i$ в алгоритме Mersenne Twister рассматриваются как $w$-компонентные векторы из чисел 0 и 1 (в дальнейшем — векторы над полем $F_2$), представляющие собой побитовое представление целых чисел $w$-битной точности. Ниже $w = 32$, поскольку именно это значение соответствует одинарной точности вычислений на графических процессорах.

Рекуррентное соотношение для генератора Mersenne Twister имеет вид

$$x_{k+n} = x_{k+m} + \left(x_k^{upper} \mid x_{k+1}^{lower}\right) \bullet \vec{A},$$

где $x_k^{upper} \mid x_{k+1}^{lower}$ — соединение $r$ первых битов вектора $x_k$ с $(w–r)$ последними битами вектора $x_{k+1}$. Здесь $r$ — параметр в диапазоне от 1 до $w–1$; $\bullet$ — операция скалярного умножения вектора $x = x_k^{upper} \mid x_{k+1}^{lower}$ справа на матрицу $\vec{A}$, при котором нули и единицы складываются по модулю 2 (операция XOR); $\vec{A}$ — произвольная двоичная матрица размерности $(w \times w)$ над полем $F_2$.

Для того чтобы умножение на матрицу $\vec{A}$ могло быть представлено простой серией побитовых операций над числом $x$, эта матрица выбирается в виде

$$\vec{A} = \begin{Vmatrix} 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & \dots \\ \dots & \dots & \dots & \dots \\ a_{w-1} & a_{w-2} & \dots & a_0 \end{Vmatrix},$$

где $a_{w–1}$, $a_{w–2}$, ..., $a_0$ — вектор над полем $F_2$, представляющий некоторое целое число с побитовым представлением $a_{w–1}$, $a_{w–2}$, ..., $a_0$. В таком случае умножение $\vec{x} \bullet \vec{A}$ реализуется следующими побитовыми операциями

$$x \bullet A = y = \begin{cases} (x >> 1) \oplus a & \text{при } x_0 = 1; \\ (x >> 1) & \text{при } x_0 = 0, \end{cases} \tag{1.3}$$

где $(x >> n)$ — операция побитового сдвига числа $x$ вправо на $n$ битов; $\oplus$ — сложение по модулю 2 (побитовая операция XOR). Для улучшения свойств случайного распределения, полученное число $y$ умножается (операцией «•») на «закаливающую» (англ. *tempering*) матрицу $T$, такую, что это умножение представляется следующей последовательностью побитовых операций

$$z = y \oplus (z >> u); \; z = z \oplus (z << s) \text{ AND } b;$$
$$z = z \oplus (z << t) \text{ AND } c; \tag{1.4}$$
$$z = z \oplus (z >> l),$$

где $u, s, t, l, b, c$ — параметры генератора, представляющие собой 32-битные векторы над полем $F_2$, соответствующие некоторым целым числам $w$-битной точности. В результате всех операций очередной элемент псевдослучайной последовательности $x_k$ ($k \geqslant n$) оказывается функцией трех предыдущих элементов

$$x_k = f\left(x_{k-n}, x_{k-n+1}, x_{k-n+m}\right).$$

Значения $m$ и $n$, вместе с другими константами, входящими в формулы (1.3)– (1.4), являются параметрами генератора. Первые $n$ членов последовательности $\{x_0, x_1, \ldots, x_{n-1}\}$ должны быть заданы как инициализирующая последовательность (англ. *seeds*).

Для перехода от случайной последовательности целых чисел $\{x_i\}$ к последовательности вещественных чисел $\gamma \in [0; 1]$ достаточно разделить все числа $x_i$ на $2^w$. Если $w = 32$, то

$$\gamma_i = x_i / 2^w = x_i / 2^{32} = x_i / 4\,294\,967\,296.$$

**Период генератора и значения параметров алгоритма.** Согласно работе [12] период генератора Mersenne Twister — количество случайных чисел, генерируемых до зацикливания последовательности, — при оптимальных сочетаниях параметров дается по формуле

$$T = 2^{nw-r} - 1,$$

где $n$ — длина инициализирующей последовательности $\{x_0, x_1, ..., x_{n-1}\}$; $w$ и $r$ — параметры алгоритма, рассмотренные выше. Для получения такого периода и максимального качества случайной последовательности, остальные параметры генератора ($a$, $b$, $c$, $u$, $s$, $t$, $l$) должны быть согласованы со значениями $n$, $w$ и $r$.

В работе [12] были исследованы различные наборы параметров, как лучший из них по качеству равномерного распределения был принят набор $n = 624$, $m = 397$, $r = 31$, $a =$ h9908B0DF, $b =$ h9D2C5680, $c =$ hEFC6000, $u = 11$, $s = 7$, $t = 15$, $l = 18$. Этим значениям соответствует период

$$T = 2^{19937} - 1 \approx 10^{6002} \text{ чисел},$$

более чем достаточный для любого практического приложения. Этот период является максимальным среди всех генераторов.

### 1.2.5. Достоинства генератора Mersenne Twister

**Производительность и ресурсоемкость алгоритма.** В результате сравнения алгоритма Mersenne Twister с другими популярными генераторами псевдослучайных чисел было выявлено, что Mersenne Twister входит в число наиболее быстрых генераторов, производительности которых примерно одинаковы. Объем памяти, требуемый для генератора Mersenne Twister, определяется длиной инициализирующей последовательности $\{x_0, x_1, ..., x_{n-1}\}$. При максимальном качестве случайного распределения значение $n$ равнялось 624 числам, что много по сравнению с большинством других генераторов [12]. Однако при необходимости экономного использования памяти значение $n$ можно уменьшить по крайней мере в 30 раз без принципиального ухудшения качества генератора [15].

**Качество случайной последовательности.** Одной из важных характеристик псевдослучайных последовательностей, создаваемых генератором Mersenne Twister, являются очень большие периоды.

Другим необходимым требованием к генератору псевдослучайных чисел являются равномерность и некоррелированность создаваемой им последовательности. Поскольку последовательности псевдослучайных чисел не являются абсолютно случайными, гарантировать применимость генератора для решения *любой* задачи невозможно. Тем

не менее существуют тесты (см., например, [13]– [14]), позволяющие обнаружить недостатки генераторов, влияющие на точность решения известных задач. Генератор Mersenne Twister удовлетворяет всем проводившимся тестам [12], в частности, тестам *diehard* [13], *Load Tests* и *Ultimate Load Tests* [14].

Сами авторы Mersenne Twister предлагают для оценки и сравнения качества генераторов псевдослучайных чисел наглядную количественную характеристику, называемую «равномерным распределением в $k$ измерениях с $w$-битной точностью» [12]. Геометрический смысл этой характеристики состоит в следующем (рис. 1.3):

1) рассматривается $k$-мерный куб с ребрами единичной длины в каждом из измерений;
2) генерируется последовательность случайных чисел $\{\gamma_i\}$ ($\gamma \in [0; 1]$, $i = 0, 1, \ldots, P–1$) длиной $P$, равной периоду генератора;
3) для всех $i = 0, 1, \ldots, P–1$ формируется $P$ точек в $k$-мерном пространстве, таких, что координаты $i$-й точки — $(\gamma_i, \gamma_{i+1}, \ldots, \gamma_{i+k-1})$, а при $i+k > P$ вместо обычного сложения используется сложение по модулю $P$;

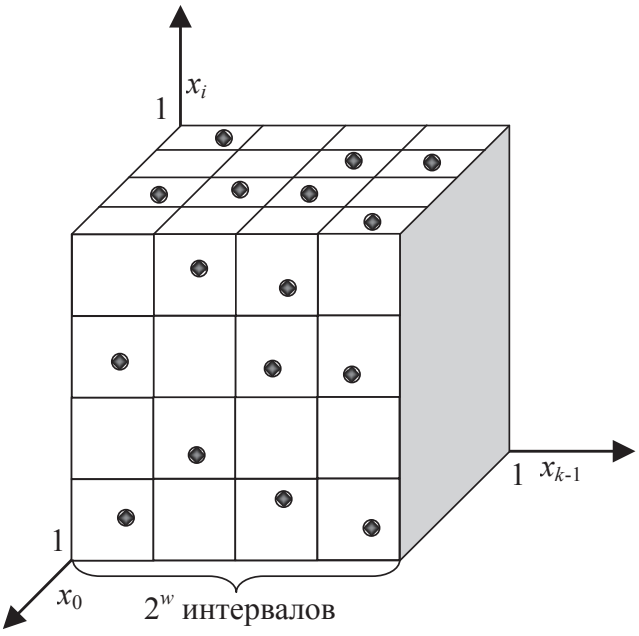



Рис. 1.3. Иллюстрация критерия равномерного распределения в $k$ измерениях с $w$-битной точностью

4) каждая ось единичного куба в $k$-мерном пространстве разбивается на $2^{\omega}$ интервала, в соответствии с чем объем куба разбивается на $2^{k\omega}$ маленьких кубика. Количество интервалов, равное $2^{\omega}$, используется потому, что при этом в один и тот же интервал попадают все числа, у которых равны первые ω-бит. Таким образом, числа сопоставляются с ω-битной точностью;
5) последовательность $\{\gamma_i\}$ является равномерно распределенной в $k$ измерениях с ω-битной точностью, если в каждый из $2^{k\omega}$ кубиков попадает одинаковое число точек, рассмотренных в п. 3 (на одну точку меньше для кубика, расположенного в начале координат).

Чем выше максимальные значения $k(\omega)$, удовлетворяющие критерию п. 5, тем более качественным можно считать генератор псевдослучайных чисел. Как показано в работе [12], для генератора Mersenne Twister максимально достижимые (при наилучших сочетаниях всех параметров) значения $k(\omega)$ описаны формулой

$$k(\omega) \le \left\lfloor \frac{nw - r}{\omega} \right\rfloor,$$

где $n$, $r$ — параметры генератора; $w$ — длина используемых целых чисел в битах (обычно $w = 32$), совпадающая с максимальной точностью представления генерируемых случайных чисел. При $n = 624$, $r = 31$ (см. пояснения в параграфе 1.2.4) и $\omega = w = 32$ получается, что $k_{\max}(\omega) = 623$. Поскольку указанные в параграфе 1.2.4 значения параметров [12] обеспечивают $k(32) = k_{\max}(32)$, генератор Mersenne Twister оказывается равномерно распределенным в 623 измерениях с 32-битной точностью.

### 1.2.6. Применение генератора Mersenne Twister для параллельных расчетов на графических процессорах

**Параллельные генераторы с различающимися параметрами.** Алгоритм Mersenne Twister может исполняться графическими процессорами архитектуры CUDA с высокой скоростью, поскольку используемая побитовая арифметика реализована в этих процессорах аппаратно. Существует проблема, связанная с параллельным запуском нескольких (тем более многих) экземпляров одного и того же генератора псевдослучайных чисел. Даже при различных «инициализаторах» $\{x_0, x_1, \ldots, x_{n-1}\}$ та-

кие генераторы могут создавать коррелирующие последовательности чисел, если имеют идентичные параметры. В частности, это справедливо и для генератора Mersenne Twister, поскольку он использует линейное рекуррентное соотношение [11].

Таким образом, экземпляры генератора Mersenne Twister, параллельно исполняемые в различных вычислительных потоках графического процессора, должны иметь различные наборы параметров ($n$, $m$, $r$, $a$, $b$, $c$, $u$, $s$, $t$, $l$). Эти параметры нельзя выбирать случайным или другим произвольным образом во избежание ухудшения качества генератора. По этой причине авторами генератора Mersenne Twister был разработан специальный алгоритм подбора параметров (DC [14]).

Алгоритм DC, в соответствии с идентификаторами вычислительных потоков, создает для каждого потока уникальные наборы параметров генератора, гарантирующие максимальные качество и период распределения. При этом 16-битное значение идентификатора потока (ID) встраивается в параметр $a$ следующим образом:

$$a = (\text{случайные 16 бит} \,|\, \text{16 бит ID}).$$

Затем алгоритм DC подбирает значения параметров $b$ и $c$, гарантирующие качественную работу генератора. Остальные параметры могут быть постоянными для всех генераторов.

Для ускорения моделирования методом Монте-Карло, наборы параметров для каждого из параллельных генераторов должны быть сгенерированы заранее и сохранены в виде массива. Это можно сделать с помощью программы на C [14]. Мы использовали реализацию этой программы на CUDA, разработанную компанией NVIDIA для расчета диффузии нейтронов через пластину [15].

**Распределение памяти GPU между параллельными генераторами.** Отметим, что для параллельного запуска многих экземпляров Mersenne Twister на графическом процессоре плохо подходят параметры со значением $n = 624$ или того же порядка величины из-за большого размера требуемой памяти. Действительно, каждому экземпляру генератора потребуется память для трех уникальных 32-битных параметров ($a$, $b$, $c$) и $n$ 32-битных элементов инициализирующей последовательности $\{x_0, x_1, \ldots, x_{n-1}\}$. Все эти значения необходимо хранить в регистрах GPU для быстрого доступа.

Ниже мы рассмотрим подробности архитектуры графических процессоров, совместимых с CUDA. Сейчас отметим только, что эти GPU

состоят из нескольких (на сегодня от 1 до 30) мультипроцессоров, каждый из которых имеет 8192 либо 16 384 доступных программисту 32-битных регистра. Таким образом, на каждом мультипроцессоре можно будет запустить не более 13 (при 8192 регистрах) либо 26 (при 16 384 регистрах) параллельных генераторов с $n = 624$, а этого мало для полного использования ресурсов GPU. Нужны параметры с намного меньшими значениями $n$.

В примере параллельного генератора Mersenne Twister для CUDA компанией NVIDIA предложен набор параметров $\{n = 19, m = 9, r = 1, u = 12, s = 7, t = 15, l = 18\}$ [15]. При $n = 19$ на каждом мультипроцессоре можно запустить 431 либо 862 параллельных генератора, что достаточно для полной загрузки GPU. Указанные параметры сохраняют высокое качество получаемых равномерных распределений, поскольку при $n = 19$, $m = 9$ и $r = 1$ генератор по-прежнему имеет большой период, равный $2^{607}-1 \approx 10^{183}$, а также характеризуется равномерным распределением с 32-битной точностью в 18 измерениях.

# 2. Задача о диффузии нейтронов через пластину. Решение методом Монте-Карло

## 2.1. Физическая формулировка задачи

### 2.1.1. Распараллеливание независимых вычислений

В качестве примера реализации метода Монте-Карло на CUDA рассмотрим задачу о диффузии нейтронов через пластину. Пусть на однородную пластину $0 \le x \le h$ перпендикулярно поверхности падает моноэнергетический поток нейтронов (рис. 2.1). В направлениях, перпендикулярных падающему потоку, будем считать пластину бесконечной. Отдельный нейтрон может отразиться от пластины, поглотиться в ней либо пройти через пластину. Целью моделирования будет определение вероятностей этих исходов.

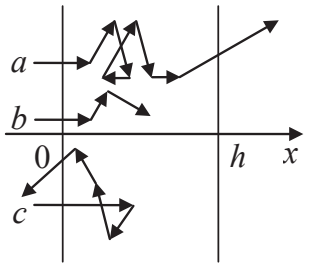



Рис. 2.1. Варианты движения нейтронов через пластину:
*a* — прохождение; *b* — поглощение; *c* — отражение

Моделирование будет состоять в отслеживании траекторий большого количества нейтронов до поглощения либо выхода из пластины. Для этого потребуется определять проекции траекторий нейтронов на ось $x$ после каждого столкновения с ядром, а также учитывать поглощения.

Вероятность столкновения нейтрона с ядром в веществе пластины определяется макросечением взаимодействия $\Sigma = N\sigma$, равным произведению числовой концентрации ядер $N$ и микросечения $\sigma$, представляющего собой эффективную площадь ядра как мишени для нейтрона. Средняя длина свободного пробега нейтрона в пластине дается по формуле $\langle\lambda\rangle = 1/\Sigma$, но конкретные значения $\lambda$ между двумя последовательными являются случайными. Известно, что как случайная величина $\lambda$ имеет плотность распределения

$$w_\lambda(x) = \Sigma \exp[-\Sigma x].$$

При строгом решении микросечение взаимодействия $\sigma$ и макросечение $\Sigma$ должны зависеть от скорости сближения нейтрона с ядром. Для упрощения моделирования мы будем считать $\Sigma$ константой, что может соответствовать таким физическим приближениям, как равенство и постоянство энергий всех нейтронов, слабая зависимость макросечения от энергии, использование макросечения, заранее усредненного по энергиям.

Исход взаимодействия нейтрона с ядром определяется вероятностями рассеяния и поглощения. Эти вероятности пропорциональны микросечениям рассеяния и поглощения $\sigma_s$ и $\sigma_a$, которые в сумме дают полное микросечение $\sigma$. Таким образом, вероятности рассеяния и поглощения нейтрона при столкновении $P_s$ и $P_a$ определяются по формулам

$$P_s = \frac{\sigma_s}{\sigma} = \frac{\Sigma_s}{\Sigma}, \quad P_a = \frac{\sigma_a}{\sigma} = \frac{\Sigma_a}{\Sigma},$$

где $\Sigma_s$, $\Sigma_a$ — макросечения рассеяния и поглощения, $\Sigma_s = N\sigma_s$ и $\Sigma_a = N\sigma_a$. Эти величины аналогично полному макросечению $\Sigma$ будем считать константами. Три указанных макросечения связаны соотношением $\Sigma = \Sigma_s + \Sigma_a$.

### 2.1.2. Рассеяние нейтронов на ядрах

Если нейтрон при столкновении с ядром не поглощается, то он испытывает рассеяние — изменяет направление движения. Поскольку мы не описываем каждое из столкновений на уровне ядерных расстояний и сил, угол отклонения нейтрона после столкновения нужно считать случайной величиной.

Пусть нейтрон и ядро взаимодействуют как две свободные частицы (что для ядра является приближением из-за взаимодействия с электронами и другими ядрами вещества). В таком случае столкновение наиболее просто выглядит в системе отсчета, связанной с центром масс нейтрона и ядра, где центр масс неподвижен. Схема отклонения нейтрона от неподвижного центра масс показана на рис. 2.2.

Опыт показывает, что в системе центра масс рассеяние нейтрона на ядре можно считать сферически симметричным. При этом все направления вектора скорости нейтрона после столкновения равнове-

роятны, а косинус μ угла между этим вектором и любой осью равномерно распределен в диапазоне [–1; 1].

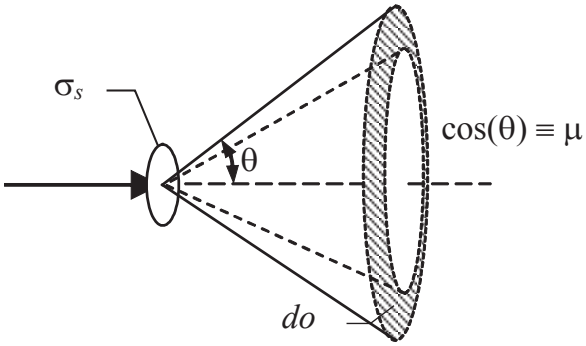



Рис. 2.2. Рассеяние нейтрона на ядре

При переходе от системы отсчета, связанной с центром масс нейтрона и ядра, к неподвижной системе, которая связана с пластиной, сферически симметричное рассеяние искажается, поскольку случайный «новый» вектор скорости складывается с исходной скоростью центра масс системы нейтрон–ядро. Этот эффект известен как «рассеяние вперед», и он заметно проявляется на легких ядрах.

В настоящем примере мы пренебрегаем «рассеянием вперед» для упрощения алгоритма, поскольку основной задачей является не максимально точное моделирование, а сравнение производительностей центрального и графического процессоров. Таким образом, распределение направлений скорости нейтрона после любого столкновения считается сферически симметричным в неподвижной системе отсчета. Это означает, что косинус угла между вектором скорости нейтрона и осью $x$ (рис. 2.1) после очередного столкновения принимает новое случайное значение μ, равномерно распределенное на отрезке [–1; 1].

Для расчета проницаемости пластины, бесконечной в двух направлениях, перпендикулярных оси $x$, достаточно отслеживать перемещения нейтронов только вдоль этой оси. После $i$-го столкновения $x$-координата нейтрона изменится в соответствии с уравнением

$$x_{i+1} = x_i + \lambda_i \mu_i, \tag{2.1}$$

где $\lambda_i$ — случайное значение длины свободного пробега после $i$-го столкновения; $\mu_i$ — случайный косинус отклонения траектории от оси $x$ (рис. 2.3).

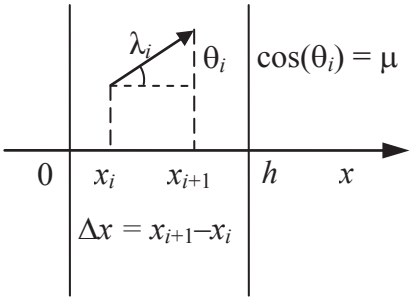



Рис. 2.3. Вычисление абсциссы следующего столкновения нейтрона с ядром

## 2.2. Численное решение задачи методом Монте-Карло

### 2.2.1. Принцип моделирования

В сформулированной модели результат прохождения конкретного нейтрона через пластину определяется случайными величинами — длинами свободного пробега $\lambda$, косинусами углов отклонения $\mu$, а также событиями поглощения либо рассеяния. Распределения нужных случайных величин известны, так что их конкретные значения могут быть получены с помощью генератора случайных чисел. Таким образом, могут быть построены траектории отдельных нейтронов. Усреднение результатов для большого количества нейтронов позволит определить вероятности прохождения пластины, отражения и поглощения. В этом и состоит решение задачи методом Монте-Карло.

### 2.2.2. Получение необходимых случайных величин

Моделирование движения отдельного нейтрона через пластину методом Монте-Карло состоит в определении его перемещения вдоль оси $x$ после очередного столкновения и «разыгрывании» случайной координаты следующего столкновения. Эти операции повторяются, пока нейтрон не выходит за пределы пластины.

Как уже отмечено выше, длина свободного пробега нейтрона $\lambda$ является случайной величиной с плотностью распределения $w\lambda(x) =$

$= \Sigma \exp[-\Sigma x]$. Этой плотности соответствует интегральная функция распределения

$$W_\lambda(x) = \int_0^x w_\lambda(x')dx' = 1 - \exp[-\Sigma x].$$

Можно показать, что если $\gamma$ — случайная величина, равномерно распределенная на отрезке [0; 1], то случайная величина $\xi \in (a; b)$, удовлетворяющая соотношению

$$W(\xi) \equiv \int_a^\xi w(x)dx = \gamma,$$

имеет интегральную функцию распределения $W(x)$ и плотность распределения $w(x)$. Таким образом, при наличии генератора случайных чисел, позволяющего получать равномерно распределенные значения $\gamma$, случайные значения $\lambda$ можно разыгрывать по формулам

$$\gamma = 1 - \exp[-\Sigma\lambda] \Rightarrow$$

$$\lambda = -\frac{1}{\Sigma}\ln(1-\gamma).$$

Если $\gamma$ — случайная величина, равномерно распределенная на отрезке [0; 1], то и величина $(1-\gamma)$ имеет такое же распределение

$$\lambda = -\frac{1}{\Sigma}\ln(\gamma). \qquad (2.2)$$

Для получения проницаемости пластины достаточно рассчитывать проекции перемещения нейтронов вдоль оси $x$ (рис. 2.3), которые определяются косинусом угла между траекторией нейтрона и этой осью (рис. 2.2). Если случайное значение этого косинуса равно $\mu$, то перемещение нейтрона вдоль оси $x$ до следующего столкновения дается по формуле $\Delta x = \lambda\mu$.

В используемом приближении без «рассеяния вперед» случайная величина $\mu$ равномерно распределена на отрезке $[-1; 1]$, чему соответствует плотность распределения

$$w_\mu(x) = \begin{cases} 1/2 \text{ при } -1 \le x \le 1; \\ 0 \quad \text{при других } x. \end{cases}$$

Связь со стандартной случайной величиной $\gamma$, равномерно распределенной на отрезке [0; 1], следует из формулы (1.1)

$$\int_{-1}^{\mu} \frac{dx}{2} = \frac{\mu - 1}{2} = \gamma \Rightarrow \mu = 2\gamma - 1 . \qquad (2.3)$$

Вероятность рассеяния при каждом столкновении равна $\Sigma s/\Sigma$. Разыграть рассеяние либо поглощение нейтрона с помощью стандартной случайной величины $\gamma \in [0; 1]$ можно, считая, что нейтрон рассеивается при

$$\gamma \le \frac{\sum_s}{\sum} \qquad (2.4)$$

и поглощается в противоположном случае.

### 2.2.3. Построение траекторий отдельных нейтронов

Для получения стандартных случайных последовательностей $\{\gamma i\}$ будем использовать рассмотренный выше генератор псевдослучайных чисел Mersenne Twister. Алгоритм моделирования заключается в следующем:

1) задать начальное положение нейтрона $x_0 = 0$ и косинус начального отклонения траектории от оси $x$ $\mu_0 = 1$;
2) сгенерировать длину свободного пробега нейтрона до первого столкновения $\lambda_1$ по формуле (2.2);
3) вычислить новую координату нейтрона на оси $x$ по формуле (2.1):

$$x_i = x_{i-1} + \lambda_i \mu_{i-1},$$

   где $i$ — номер столкновения;
4) если $x_i > h$, то завершить расчет, считая нейтрон прошедшим через пластину;
5) если $x_i < 0$ (что возможно при $i > 1$), то завершить расчет, считая нейтрон отразившимся от пластины;
6) если $0 \le x_i \le h$, то разыграть исход столкновения — рассеяние либо поглощение — по формуле (2.4). В случае поглощения нейтрона закончить расчет;
7) в случае рассеяния сгенерировать новый косинус угла отклонения траектории $\mu_i$ по формуле (2.3), после чего продолжить расчет начиная с п. 2.

Для получения окончательного результата необходимо провести моделирование для многих нейтронов. Пусть $N_+$ — количество нейтронов, прошедших сквозь пластину, $N_-$ — количество отразившихся нейтронов, $N_a$ — количество поглощенных нейтронов. В таком случае проницаемость пластины совпадает с вероятностью ее прохождения

$$P_+ = \frac{N_+}{N_+ + N_- + N_a},$$

а коэффициент отражения — с вероятностью отражения

$$P_- = \frac{N_-}{N_+ + N_- + N_a}.$$

Соответственно этому вероятность поглощения дается по формуле

$$P_a = 1 - P_+ - P_- = \frac{N_a}{N_+ + N_- + N_a}.$$

### 2.2.4. Алгоритм с использованием «веса» нейтронов

В алгоритме, изложенном выше, поглощаемый нейтрон исчезает полностью, а вычисления, проведенные для его траектории, в дальнейшем не учитываются. Получается, что эти расчеты не в полной мере используются при получении результатов, что снижает эффективность алгоритма. Потери промежуточных расчетов можно избежать, если моделирование всех траекторий будет продолжаться от входа в пластину и до выхода из нее, без обрыва при поглощениях. Для этого подходит следующая модификация алгоритма, включающая «вес» нейтронов.

Пусть по каждой из моделируемых траекторий движется не один нейтрон, а целое множество, которому в реальности может соответствовать множество нейтронов на похожих траекториях. При каждом столкновении количество этих нейтронов будет уменьшаться на величину, пропорциональную вероятности поглощения $\Sigma_a/\Sigma$, но часть нейтронов дойдет до выхода из пластины. Можно не рассматривать все эти нейтроны напрямую, а ввести «вес» нейтрона $\omega$, равный отношению количества нейтронов, остающихся на траектории после очередного столкновения, к исходному количеству. До первого столкновения $\omega_0 = 1$. При каждом $i$-м столкновении вес уменьшается по формуле

$$\omega_{i+1} = \omega_i \left(1 - \Sigma_a / \Sigma\right), \tag{2.5}$$

но моделирование траектории не может закончиться из-за поглощения.

Преимуществами алгоритма с «весом» являются: использование всех рассчитанных траекторий до самого конца моделирования — можно показать, что при моделировании того же количества траекторий точность алгоритма с «весом» выше, чем без него [2]; отсутствие необходимости получения случайных чисел для выбора между поглощением либо рассеянием нейтрона — уменьшение количества обращений к генератору случайных чисел увеличивает скорость моделирования.

Алгоритм, с помощью которого определяют «вес» нейтронов, включает следующие этапы:

1) наряду с начальной координатой $x_0 = 0$ и косинусом отклонения траектории $\mu_0 = 1$, задается также вес нейтрона $\omega_0 = 1$;
2) длины свободного пробега нейтрона на каждом шаге генерируются так же, как в предыдущем алгоритме, по формуле (2.2);
3) новые координаты нейтронов $x_i = x_{i-1} + \lambda_i \mu_{i-1}$ по-прежнему вычисляются по формуле (2.1);
4) если $x_i > h$, то расчет завершается без модификации веса. То же самое и при $x_i < 0$;
5) если $0 \le x_i \le h$, то модифицируют вес по формуле (2.5). Расчет не прекращается;
6) генерируют новый косинус угла отклонения траектории $\mu_i$ по формуле (2.3), после чего продолжают расчет начиная с п. 2.

Результат моделирования теперь определяется по следующим формулам. Пусть $N_+$ — количество нейтронов с «весом», прошедших сквозь пластину, а $N_-$ — количество отразившихся нейтронов с «весом». Полностью поглощенных нейтронов теперь нет, так что полное количество нейтронов $N = N_+ + N_-$. Вероятность прохождения пластины пропорциональна сумме конечных весов всех прошедших нейтронов

$$P_+ = \frac{1}{N_+ + N_-} \sum_{i=1}^{N_+} \omega_i \ , \tag{2.6}$$

а вероятность отражения от пластины — сумме весов всех отразившихся нейтронов

$$P_- = \frac{1}{N_+ + N_-} \sum_{i=1}^{N_-} \omega_i \ . \tag{2.7}$$

Вероятность поглощения можно найти по формуле

$$P_a = 1 - P_+ - P_-, \tag{2.8}$$

что эквивалентно соотношению

$$P_a = \frac{1}{N_+ + N_-} \sum_{i=1}^{N} (1 - \omega_i), \tag{2.9}$$

в котором суммирование ведется по всем нейтронам, как прошедшим пластину, так и отразившимся.

# 3. Решение задачи о прохождении нейтронов через пластину на графических процессорах архитектуры CUDA

## 3.1. Графический процессор как система для параллельных вычислений общего назначения

### 3.1.1. Архитектура графического процессора

Особенности графических процессоров как систем для параллельных вычислений общего назначения детально изложены, например, в описании [4]. Здесь мы ограничимся кратким обсуждением основных принципов распараллеливания вычислений на графических процессорах.

Графический процессор (GPU) представляет собой отдельную вычислительную микросхему, работающую параллельно с центральным процессором. На персональных компьютерах GPU обычно входят в состав графических ускорителей (или видеокарт) — дополнительных устройств, изначально предназначенных для визуализации двух- и трехмерной графики. Графические ускорители, помимо GPU, включают в себя микросхемы видеопамяти (см. ниже) и собственную систему охлаждения. Они подключаются к системной плате через шины данных PCI-Express либо AGP, которые обеспечивают центральному процессору доступ к видеопамяти и GPU.

На рис. 3.1 в качестве примера показана архитектура графического процессора G80– одного из процессоров NVIDIA, поддерживающих архитектуру CUDA. Как и другие, этот GPU представляет собой систему из параллельных вычислительных блоков (потоковых мультипроцессоров), каждый из которых исполняет одну и ту же программу (англ. *kernel* — вычислительное ядро) применительно к различным исходным данным.

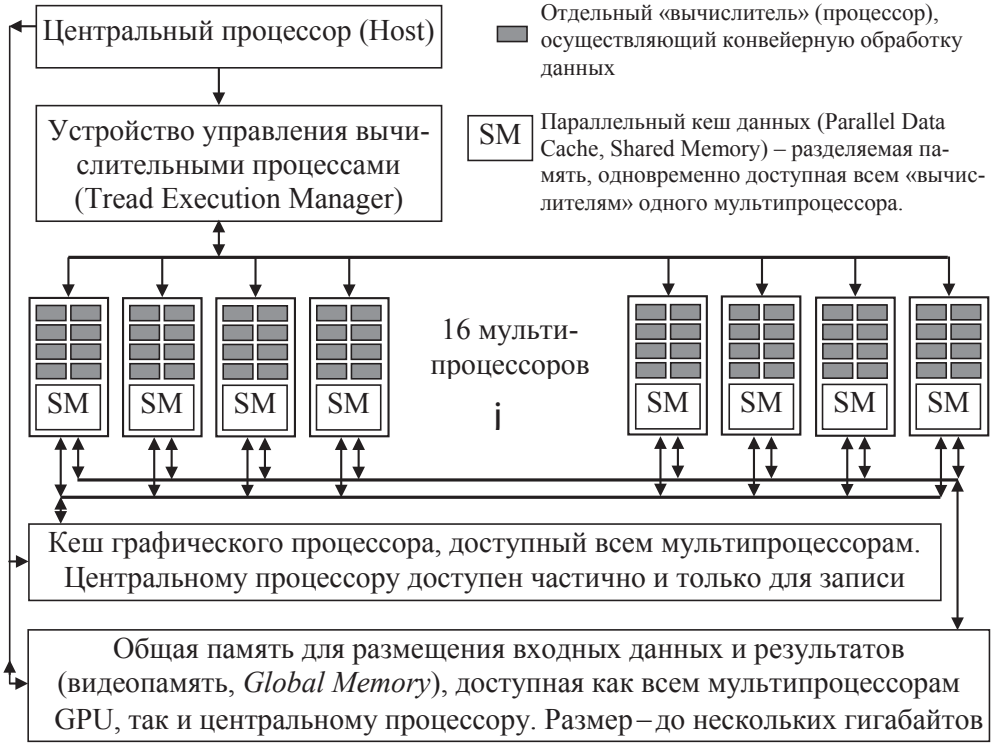



Рис. 3.1. Архитектура графического процессора NVIDIA G80 [1]

Параллельная архитектура графических процессоров ориентирована на исполнение алгоритмов, в которых одна и та же последовательность операций должна быть многократно исполнена либо для обработки элементов больших массивов данных (распараллеливание вычислений по данным), либо для моделирования одной и той же системы с различными параметрами (один из вариантов распараллеливания по задачам).

В настоящем пособии мы рассмотрим только процессоры архитектуры CUDA, выпускаемые компанией NVIDIA, как наиболее удобные для неграфических вычислений. Название CUDA — Compute Unified Device Architecture (архитектура универсального вычислительного устройства) — подчеркивает тот факт, что эти процессоры предназначены не только для обработки графики, но и для выполнения алгоритмов общего назначения.

Концепцию вычислений, реализованную в архитектуре CUDA, NVIDIA характеризует понятием *SIMT — Single Instruction for Multiple*

*Threads* (одна инструкция для множества потоков). Это название, модифицированное от известной аббревиатуры *SIMD* (*Single Instruction for Multiple Data*, одна инструкция для множества данных), подчеркивает тот факт, что конкретные вычислительные потоки в CUDA не «привязаны» к конкретным элементам массивов исходных данных. Напротив, потоки хорошо управляемы, имеют уникальные идентификаторы, доступные программисту, осуществляют произвольный доступ к данным из памяти, посредством разделяемой памяти могут взаимодействовать друг с другом.

### 3.1.2. Возможности программирования процессоров архитектуры CUDA

Программирование GPU на CUDA мы рассмотрим на примере задачи о прохождении нейтронов через пластину. Общие принципы такого программирования заключаются в следующем.

Язык программирования на CUDA представляет собой расширение языка C. Его компилятор [5] поддерживает все стандартные операторы и операции языка C (для вычислительных ядер в пределах технических возможностей графического процессора), все широко используемые математические функции, а также специфические для CUDA операции, предназначенные для управления памятью GPU и для организации параллельных вычислений. Этот компилятор может быть подключен к таким средам программирования, как Microsoft Visual Studio 2005–2013, Microsoft Visual C++ Express 2005–2013.

Современные GPU архитектуры CUDA могут работать с вещественными числами 32-битной (одинарной) и 64-битной (двойной) точности, а также с 32- и 64-битными целыми числами. Поддерживаются побитовые логические операции с целыми числами, необходимые, в частности, для высокоскоростной реализации генераторов псевдослучайных чисел, требуемых для метода Монте-Карло.

Графическому процессору доступна память нескольких типов [4], из которых для вычислений общего назначения особенно важны следующие:

- регистры графического процессора. Расположены прямо на кристалле GPU, так что доступ к ним (чтение либо запись числа 32-битной точности) требует минимально возможного времени —

4 тактов процессора. Каждый из потоковых мультипроцессоров в зависимости от версии CUDA [4] имеет 8192 либо 16 384 32-битных регистра. Регистры доступны центральному процессору для записи данных перед началом исполнения вычислительного ядра. Отметим, что для хранения чисел двойной (64-битной) точности используются пары этих же регистров;

- разделяемая память (*Shared Memory*). Как и регистры, расположена на кристалле GPU и работает с такой же высокой скоростью (4 такта для чтения и для записи). В отличие от регистров, распределяемых по вычислительным потокам (*threads*), доступна одновременно всем вычислительным потокам, исполняемым на одном и том же мультипроцессоре. Позволяет этим потокам взаимодействовать между собой. Имеет объем 16 Kb (16 384 байта, то есть 4096 32-битных ячеек) на мультипроцессор. Центральному процессору недоступна;
- общая память (*Global Memory*). Представляет собой отдельные микросхемы памяти, расположенные на плате графического ускорителя. Имеет большой объем — от 256 Mb до нескольких гигабайтов. Предназначена для хранения больших массивов исходных данных и для сохранения результатов работы вычислительного ядра. Доступ из вычислительного ядра требует 400–600 тактов [5], так что эта память работает на два порядка медленнее, чем регистры. Доступна одновременно всем мультипроцессорам GPU и всем вычислительным потокам, а также центральному процессору как для записи, так и для чтения. Через эту память центральный процессор получает результаты вычислений на GPU;
- также кеш графического процессора для констант (*Constant cache*) и текстур (*Texture cache*). Эти разделы памяти работают со скоростью регистров и доступны одновременно всем мультипроцессорам. Они применяются для оптимизации доступа к константам и графическим данным (текстурам). Подробности использования данных типов памяти приведены в описании [4].

Поскольку архитектура CUDA совершенствуется, возможности графических процессоров, выпускавшихся в разное время, несколько различны. Для обобщения этих возможностей компания NVIDIA использует понятие «вычислительной способности» (*Compute Capability* [5]), которая к настоящему времени имеет версии до 5.0. Конкретные модели GPU, поддерживающие CUDA, могут различаться также ко-

личеством мультипроцессоров, пропускной способностью шины данных, поддерживаемыми частотами памяти и ядра.

### 3.1.3. Конвейерная обработка данных в архитектуре CUDA

Во всех существующих графических процессорах архитектуры CUDA одинаково реализована конвейерная обработка данных. Каждый из мультипроцессоров (*Stream Multiprocessor*), образующих GPU, имеет в своем составе 8 параллельных вычислительных устройств — скалярных процессоров (*Scalar Processors*). При этом он может одновременно исполнять не 8, а 1024 (или 768) параллельных вычислительных потока (по 512 в двух логических «связках»). Увеличение количества потока способствует увеличению производительности, так как потоковые процессоры имеют конвейерную архитектуру. Каждый из скалярных процессоров одновременно исполняет несколько вычислительных потоков, находящихся на разных стадиях алгоритма. Например, один процесс записывает данные в видеопамять, другой обращается к разделяемой памяти, третий — к регистрам, а еще один ведет вычисления.

Архитектура CUDA такова, что вычислительные потоки запускаются на исполнение группами (*warps*) максимально по 32 потока в каждой. Графический процессор автоматически распределяет задачи между этими группами так, чтобы увеличить количество потоков, исполняемых одновременно (на различных участках алгоритма). Чтобы максимально использовать ресурсы графического процессора, исполняемая программа должна по возможности удовлетворять следующим требованиям:

- запускать не менее двух «связок» на каждом мультипроцессоре (например, GPU GeForce 8800 (G 80) имеет 16 мультипроцессоров, так что «связок» должно быть не меньше, чем 32);
- запускать в каждой связке не менее 32-х, а лучше — 64-х потоков; количество потоков по возможности должно быть кратно 32;
- не допускать значительного расхождения потоков внутри одной группы (*warp*) при ветвлениях алгоритма.

Количества «связок» и потоков, запускаемых графическим процессором, определяются программистом в коде программы при вызове вычислительного ядра. Выбирая количество «связок» и потоков, необ-

ходимо следить за тем, чтобы всем потокам хватало регистров и разделяемой памяти, так как в противном случае компилятор CUDA будет использовать общую память (видеопамять), что резко понизит скорость вычислений.

## 3.2. Простейшее решение задачи о диффузии нейтронов через пластину на графическом процессоре

### 3.2.1. Структура программы

Приложения, использующие графический процессор, в настоящее время всегда состоят по крайней мере из двух частей. Одна часть исполняется на центральном процессоре и обеспечивает:

- взаимодействие приложения с периферийными устройствами (загрузка файлов с жесткого диска) и пользователем (запуск приложения, вывод данных на экран);
- размещение исходных данных и результатов в оперативной памяти компьютера;
- проведение непараллельных расчетов;
- копирование необходимых данных в видеопамять, доступную графическому процессору, получение из видеопамяти результатов расчета;
- запуск вычислительного ядра на графическом процессоре.

Наш пример по расчету проницаемости пластины методом Монте-Карло будет включать в себя действия, показанные на схемах рис. 3.2 и рис. 3.3.

Действия, перечисленные на схеме рис. 3.2, выполняются последовательно, одним центральным процессором. Напротив, вычислительное ядро, запускаемое на графическом процессоре, исполняется параллельно большим количеством вычислительных потоков (в нашем примере их будет $32 \cdot 128 = 4096$). Порядок работы вычислительного ядра показан на рис. 3.3.

| Центральный процессор: |
|---|
| Установка значений констант, определяющих:<br>• характеристики взаимодействия нейтронов с веществом пластины<br>• количество «связок» и вычислительных потоков, запускаемых на графическом процессоре<br>• общие параметры генератора случайных чисел |
| Загрузка (из файла) массива с параметрами, необходимыми для независимой инициализации всех параллельных генераторов случайных чисел |
| Копирование этого массива с параметрами параллельных генераторов в видеопамять, доступную графическому процессору |
| Создание в оперативной памяти компьютера массива для сохранения результатов работы графического процессора |
| Запуск вычислительного ядра на графическом процессоре |
| После окончания работы графического процессора копирование результатов расчета (данных о количестве прошедших через пластину и отразившихся нейтронах, их весах) из видеопамяти в оперативную память компьютера |
| Окончательная обработка результатов моделирования и вывод результатов на экран |

Рис. 3.2. Схема и последовательность работы центрального процессора при моделировании прохождения нейтронов через пластину

| Графический процессор, блок-схема программы, исполняемой каждым из вычислительных потоков: |
|---|
| 1. Инициализация генератора случайных чисел с использованием массива, загруженного центральным процессором в видеопамять |
| 2. Присвоение начальных значений координате, весу и углу отклонения нейтрона |
| 3. Получение случайного значения длины свободного пробега нейтрона |
| 4. Расчет новой координаты нейтрона |
| 5. Проверка условий выхода нейтрона за пределы пластины |
| 6. Окончание расчета и переход к п. 9 в случае, если нейтрон покинул пластину |
| 7. Модификация веса нейтрона по формуле |
| 8. Получение случайного значения нового отклонения траектории нейтрона от оси *x*. Переход к п. 3. |
| 9. Сохранение в видеопамяти результата моделирования, а именно:<br>• некоторое значение, показывающее, прошел нейтрон через пластину или отразился (например, 1 для прохождения и 0 для отражения);<br>• конечный вес нейтрона |

Рис. 3.3. Схема и последовательность работы центрального процессора при моделировании прохождения нейтронов через пластину

Часть программы, исполняемую на графическом процессоре, называют *вычислительным ядром*. Одно и то же вычислительное ядро одновременно выполняется всеми вычислительными потоками. Оно может включать в себя:

- получение данных из видеопамяти и запись данных в видеопамять;
- обмен данными с разделяемой памятью;
- использование регистров графического процессора;
- арифметические и логические операции, другие встроенные в CUDA математические функции;
- вызовы пользовательских функций (за исключением рекурсивных);
- ветвления программы (операторы условного перехода);
- циклы.

### 3.2.2. Текст программы для центрального процессора

Ниже с комментариями приведен текст той части программы, которая предназначена для центрального процессора. Эта программа содержит операторы, специфические для CUDA, необходимые для взаимодействия центрального и графического процессоров. Она транслируется компилятором CUDA.

Прототипом программы был пример реализации генератора случайных чисел *Mersenne Twister*, предложенный компанией NVIDIA в дистрибутиве CUDA [15]. Этот текст был модифицирован в соответствии с решаемой задачей о проницаемости пластины.

```
/* Подключение стандартных библиотек, необходимых компилятору CUDA. Осуществляется по аналогии с примером [15] из дистрибутива CUDA */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <cutil.h>
/* Подключение файлов, которые содержат:
• значения общих параметров генератора Mersenne Twister (MersenneTwister.h);
```

• процедуры для инициализации параллельных генераторов случайных чисел (MT_Initialization.cu);

• текст вычислительного ядра (Monte_Carlo_kernel.cu) */

**#include "MersenneTwister.h"**

**#include "MT_Initialization.cu"**

**#include "Monte_Carlo_kernel.cu"**

/* Произвольная константа, участвующая в инициализации генераторов случайных чисел. В данном примере она одинакова для всех генераторов, но можно задавать ее для каждого генератора случайным образом */

**const unsigned int SEED = 777;**

/* Установка количества вычислительных потоков, каждый из которых будет моделировать движение одного нейтрона «с весом» через пластину */

**const int Number_of_Threads = 32 * 128;//= 4096**

/* Определение объема памяти, необходимого для сохранения результатов расчета: 2 числа по 4 байта на каждый поток */

**const int OutPut_Memory = 2 * 4 * Number_of_Threads;**

/* Процедура, с которой начинается исполнение приложения */

**int main (int argc, char **argv)**

**{**

/* Описание указателей на массивы чисел типа float (вещественные числа одинарной точности), которые будут размещены в видеопамяти и в оперативной памяти компьютера для записи результатов моделирования */

**float *d_Rand;//Указатель для видеопамяти**

**float *h_RandGPU//Указатель для оперативной памяти;**

/* Инициализация графического процессора */

**CUT_DEVICE_INIT (argc, argv);**

/* Выделение области для хранения результатов в оперативной памяти компьютера. h_RandGPU — указатель на начало этой области, эквивалентный названию массива. Тип (float *) означает, что в массиве будут храниться вещественные числа одинарной (32-битной) точности. */

**h_RandGPU = (float *)malloc (OutPut_Memory);**

/* Выделение такой же области для результатов в видеопамяти графического процессора. На начало этой области будет указывать указатель d_Rand. */

**CUDA_SAFE_CALL (cudaMalloc ((void **)&d_Rand, OutPut_Memory));**

/* Автоматическое определение полного пути к файлу MersenneTwister.dat, в котором хранятся инициализирующие параметры для 4096 параллельных генераторов случайных чисел */

**const char *dat_path = cutFindFilePath ("MersenneTwister.dat", argv [0]);**

/* Загрузка этого файла и выполнение процедуры, создающей инициализирующие последовательности параметров для всех 4096 генераторов случайных чисел */

**loadMTGPU (dat_path);**

**seedMTGPU (SEED);**

/* Синхронизация вычислительных потоков перед запуском графического процессора на исполнение вычислительного ядра */

**CUDA_SAFE_CALL (cudaThreadSynchronize ());**

/* Запуск вычислительного ядра на GPU. Сам текст процедуры, названной GPU_Monte_Carlo, находится в отдельном файле «Monte_Carlo_kernel.cu», который будет рассмотрен ниже, в параграфе 3.2.3. Инструкция <<<32, 128>>> означает, что будет запущено 32 «связки» по 128 потоков в каждой. Если программа будет исполняться на процессоре G 80 или предыдущих версиях, то на каждый мультипроцессор придется не менее двух «связок». Количество потоков 128 кратно 64, что является одним из условий оптимизации вычислений (см. п. 3.1.3).*/

**GPU_Monte_Carlo<<<32, 128>>> (d_Rand);**

/* Проверка успешности исполнения вычислительного ядра на GPU и вывод на экран сообщения в случае ошибки */

**CUT_CHECK_ERROR ("RandomGPU () execution failed\n");**

/* Синхронизация вычислительных потоков после завершения расчета на графическом процессоре */

**CUDA_SAFE_CALL (cudaThreadSynchronize ());**

/* Копирование результатов моделирования из видеопамяти в оперативную память компьютера. Область видеопамяти, начало которой показывает указатель d_Rand, копируется в область оперативной памяти, на которую указывает указатель h_RandGPU */

**CUDA_SAFE_CALL (cudaMemcpy (h_RandGPU, d_Rand, OutPut_Memory, cudaMemcpyDeviceToHost));**

/* Обработка результатов моделирования. Исполняется без распараллеливания на центральном процессоре */

```
float Reflected_weight = 0.0f;/* Суммарный вес нейтронов, отразившихся от пластины/*
float Penetrating_weight = 0.0f;/* Суммарный вес нейтронов, прошедших через пластину/*
int MC_address;//Адрес очередного нейтрона в массиве
for (int i = 0; i < Number_of_Threads; i++)
{
MC_address = 2 * i;
if (h_RandGPU [MC_address] < 0.5f) {
Reflected_weight += h_RandGPU [MC_address+1];
}
if (h_RandGPU [MC_address] > 0.5f) {
Penetrating_weight += h_RandGPU [MC_address+1];
}
}//Завершение анализа массива результатов
/* С учетом формул (2.6)– (2.9) расчет и вывод на экран вероятностей прохождения пластины, отражения от пластины и поглощения в пластине*/
printf ("Probability of penetration: %E\n", Penetrating_weight/(float) Number_of_Threads);
printf ("Probability of reflection: %E\n", Reflected_weight/(float)Number_of_Threads);
printf ("Probability of absorption: %E\n", 1.0f — (Penetrating_weight + Reflected_weight)/(float)Number_of_Threads);
printf ("Probability of deflection: %E\n", (MC_weight_overall — MC_weight_through)/(float)MC_out_neutrons);
/* Освобождение видеопамяти и оперативной памяти компьютера */
CUDA_SAFE_CALL (cudaFree (d_Rand));
free (h_RandGPU);
/* Завершение работы программы */
CUT_EXIT (argc, argv);}
```

Описания пользовательских типов, а также константы, постоянного доступа к которым программисту не требуется, часто хранятся в так называемых заголовочных файлах с традиционным расширением «.h». В нашем примере используется заголовочный файл “MersenneTwister.h”, где описана структура **mt_parameters**, используемая для инициализации параллельных генераторов MersenneTwister, а также за-

даются значения постоянных для этого генератора. Текст файла “MersenneTwister.h” приведен ниже.

```
/* Заголовочный файл “MersenneTwister.h” */
#ifndef MERSENNETWISTER_H
#define MERSENNETWISTER_H
#ifndef mersennetwister_h
#define mersennetwister_h
#define DCMT_SEED 4172
#define MT_RNG_PERIOD 607
```

/* Определяемые ниже переменные фигурируют в коде программы вычислительного ядра (параграф 3.2.3). Структура mt_parameters содержит варьируемые параметры, различные для каждого из параллельных генераторов случайных чисел */

```
typedef struct
{
unsigned int matrix_a;//параметр a
unsigned int mask_b;//параметр b
unsigned int mask_c;//параметр c
unsigned int seed;/* Константа, используемая для генерации инициализирующей последовательности {x0, ..., xn-1} */
} mt_parameters;
//Параметры, одинаковые для всех параллельных генераторов
#define MT_MM 9//Параметр m
#define MT_NN 19//Параметр n
#define MT_WMASK 0xFFFFFFFFU/* Маска для получения первых 32 битов числа */
#define MT_UMASK 0xFFFFFFFEU/* Маска для получения 31 бита числа */
#define MT_LMASK 0x1U/* Маска для получения 1 последнего бита числа */
#define MT_SHIFT0 12//Параметр u
#define MT_SHIFTB 7//Параметр s
#define MT_SHIFTC 15//Параметр t
#define MT_SHIFT1 18//Параметр l
#endif
#endif
```

На центральном процессоре исполняются также процедуры loadMTGPU и seedMTGPU, первая из которых загружает файл с ини-

циализирующими параметрами для всех параллельных генераторов случайных чисел, а вторая — осуществляет инициализацию генераторов. Мы выделили эти процедуры в отдельный файл "MT_Initialization.cu", текст которого приведен ниже.

```
/* Текст файла "MT_Initialization.cu" */
/* Создание массивов для хранения инициализирующих параметров генераторов случайных чисел в видеопамяти (модификатор __device__) и оперативной памяти компьютера */
__device__ static mt_parameters device_MT [Number_of_Threads];
static mt_parameters host_MT [Number_of_Threads];
/* Процедура, загружающая параметры для инициализации генераторов случайных чисел из файла fname */
void loadMTGPU (const char *fname)
{
/* Открытие файла fname */
FILE *fd = fopen (fname, "rb");
/* Сообщение об ошибке и выход из программы в случае отсутствия файла */
if (! fd){
printf ("initMTGPU (): failed to open %s\n", fname);
exit (0);}
/* Загрузка содержимого файла в массив host_MT. Вывод сообщения и выход из программы в случае ошибки */
if (! fread (host_MT, sizeof (host_MT), 1, fd)){
printf ("initMTGPU (): failed to load %s\n", fname);
exit (0);
}
/* закрытие файла fname */
fclose (fd);}
/* Процедура, инициализирующая параллельные генераторы случайных чисел. Являющаяся аргументом переменная seed может быть либо постоянной для всех генераторов, как в данном примере, либо случайной для каждого (что улучшает взаимонезависимость генераторов) */
void seedMTGPU (unsigned int seed)
{
int i;
```

/* В оперативной памяти компьютера создается временный массив для хранения инициализирующих параметров — структур пользовательского типа mt_parameters */

**mt_parameters *MT = (mt_parameters *)malloc (Number_of_Threads * sizeof (mt_parameters));**

/* Цикл по всем Number_of_Threads = 4096 генераторам случайных чисел. Здесь они обрабатываются последовательно, так как процедура исполняется центральным процессором. Инициализирующий массив host_MT [i], загруженный из файла процедурой loadMTGPU (и модифицированный переменной seed) копируется в область памяти, на которую указывает указатель MT */

**for (i = 0; i < Number_of_Threads; i++) {**

**MT [i] = host_MT [i];**

**MT [i].seed = seed;}**

/* Область оперативной памяти, заданная указателем MT и содержащая инициализирующий массив, копируется в область видеопамяти (определяемую указателем device_MT), где она будет доступна графическому процессору */

**CUDA_SAFE_CALL (cudaMemcpyToSymbol (device_MT, MT, sizeof (host_MT)));**

/* Освобождение оперативной памяти, которую занимал массив MT */

**free (MT);}**

### 3.2.3. Текст программы для графического процессора

Для того чтобы разделить части приложения, исполняемые на центральном и графическом процессорах, текст вычислительного ядра, предназначенного для графического процессора, размещен в отдельном файле, «Monte_Carlo_kernel.cu». Этот файл подключается к предыдущему директивой #include «Monte_Carlo_kernel.cu» на этапе компиляции.

/* Вычислительное ядро. Исполняется одновременно всеми 4096 вычислительными потоками. На то что процедура является вычислительным ядром, указывает модификатор __global__ перед названием */

**__global__ void Monte_Carlo_GPU (float *d_Random)**

**{**

/* Идентификация конкретного вычислительного потока. Используются стандартные (автоматически существующие) системные переменные:

- blockDim.x — количество вычислительных потоков, приходящихся на одну «связку» (в нашем примере blockDim.x = 128);
- blockIdx.x — номер «связки», в нашем примере создаются 32 «связки»;
- threadIdx.x — номер вычислительного потока внутри «связки».

const int tid = blockDim.x * blockIdx.x + threadIdx.x;

Таким образом, tid — номер конкретного вычислительного потока, изменяющийся через все «связки» от 0 до blockDim.x * blockIdx.x — 1. */

/* Описание переменных для генератора Mersenne Twister */

**int iState, iState1, iStateM, iOut;**

**unsigned int mti, mti1, mtiM, x;**

**unsigned int mt [MT_NN];**

/* Модельные константы, описывающие взаимодействие нейтрона с пластиной: */

**const float MC_h = 1.0f;**//Толщина пластины;

**const float MC_sigma = 11.0f;**//Полное макросечение взаимодействия;

**const float MC_sigma_S = 10.0f;**//Макросечение рассеяния, //обратное средней длине свободного пробега;

**float MC_s = MC_sigma_S/MC_sigma;**//Вероятность рассеяния //нейтрона.

/* Загрузка инициализационных параметров генератора Mersenne Twister для данного конкретного вычислительного потока из device_MT, расположенного в видеопамяти; этот массив был создан процедурой seedMTGPU */

**mt_parameters config = device_MT [tid];**

/* Инициализация генератора */

**mt [0] = config.seed;**

**for (iState = 1; iState < MT_NN; iState++)**

**mt [iState] = (1812433253U * (mt [iState — 1] ^ (mt [iState — 1] >> 30)) + iState) & MT_WMASK;**

**iState = 0;**

**mti1 = mt [0];**

/* Начало генерации последовательности случайных чисел и моделирования прохождения нейтронов сквозь пластину */

**float MC_x = 0.0f;//Координата нейтрона вдоль оси x**

```
float MC_w = 1.0f/MC_s;//Начальный вес нейтрона
float MC_m = 1.0f;//Косинус угла отклонения нейтрона
float MC_L;//Длина свободного пробега
```

/* Начало цикла, каждый шаг которого соответствует одному столкновению нейтрона внутри пластины */

```
do {
```

/* Пересчет веса нейтрона в результате столкновения. Размещение этой операции в начале цикла исключает использование дополнительного условного оператора, так как при возвращении к началу цикла подразумевается столкновение. Для того чтобы в ходе первой итерации вес нейтрона равнялся 1, перед входом в цикл MC_w = 1/MC_s */

```
MC_w *= MC_s;
```

/* Первый вызов генератора Mersenne Twister для получения случайной длины свободного пробега MC_L */

```
iState1 = iState + 1;
iStateM = iState + MT_MM;
if (iState1 >= MT_NN) iState1 -= MT_NN;
if (iStateM >= MT_NN) iStateM -= MT_NN;
mti = mti1;
mti1 = mt [iState1];
mtiM = mt [iStateM];
x= (mti & MT_UMASK) | (mti1 & MT_LMASK);
x= mtiM ^ (x >> 1) ^ ((x & 1)? config.matrix_a: 0);
mt [iState] = x;
iState = iState1;
x ^= (x >> MT_SHIFT0);
x ^= (x << MT_SHIFTB) & config.mask_b;
x ^= (x << MT_SHIFTC) & config.mask_c;
x ^= (x >> MT_SHIFT1);
```

/* Преобразование случайного числа x к диапазону (0; 1) и получение случайной длины свободного пробега по формуле (2.5) */

```
MC_L = -__logf (((float)x + 1.0f)/4294967296.0f)/MC_sigma;
```

//Определение нового положения нейтрона

```
MC_x += MC_L*MC_m;
```

/* Второй вызов генератора Mersenne Twister для получения случайного отклонения новой траектории нейтрона от оси x */

```
iState1 = iState + 1;
iStateM = iState + MT_MM;
```

```
if (iState1 >= MT_NN) iState1 -= MT_NN;
if (iStateM >= MT_NN) iStateM -= MT_NN;
mti = mti1;
mti1 = mt [iState1];
mtiM = mt [iStateM];
x= (mti & MT_UMASK) | (mti1 & MT_LMASK);
x= mtiM ^ (x >> 1) ^ ((x & 1)? config.matrix_a: 0);
mt [iState] = x;
iState = iState1;
//Tempering transformation
x ^= (x >> MT_SHIFT0);
x ^= (x << MT_SHIFTB) & config.mask_b;
x ^= (x << MT_SHIFTC) & config.mask_c;
x ^= (x >> MT_SHIFT1);
```

/* Преобразование случайного числа x к диапазону (0; 1) и получение случайного косинуса угла отклонения траектории нейтрона от оси x */

```
MC_m = 2.0f* (((float)x + 1.0f)/4294967296.0f) — 1.0f;
```

/* Проверка условия выхода нейтрона из пластины и завершение цикла при выполнении этого условия */

```
} while ((MC_x < 0.0f) || (MC_x > MC_h))
```

/* Запись результатов моделирования в видеопамять. В данном примере каждый генератор пишет два числа: 0.0, если нейтрон отразился от пластины, либо 1.0, если нейтрон прошел пластину; второе число — окончательный вес нейтрона; d_Random — указатель на начало массива результатов в видеопамяти*/

```
int MC_address = 2*tid;
if (MC_x < 0.0f) d_Random [MC_address] = 0.0f;
if (MC_x > MC_h) d_Random [MC_address] = 1.0f;
d_Random [MC_address+1] = MC_w;
}//Завершение вычислительного ядра
```

### 3.2.4. Быстродействие простейшего алгоритма

Алгоритм, рассмотренный выше, хорошо демонстрирует основные идеи моделирования диффузии нейтронов через пластину методом Монте-Карло. Вместе с тем он не является оптимальным, поскольку

после завершения расчета «своей» траектории вычислительные потоки останавливаются. Эффект иллюстрируют графики (рис. 3.4–3.5), показывающие зависимость времени счета от количества нейтронов (вычислительных потоков), а также сравнение производительностей графического и центрального процессоров при выполнении этого алгоритма.

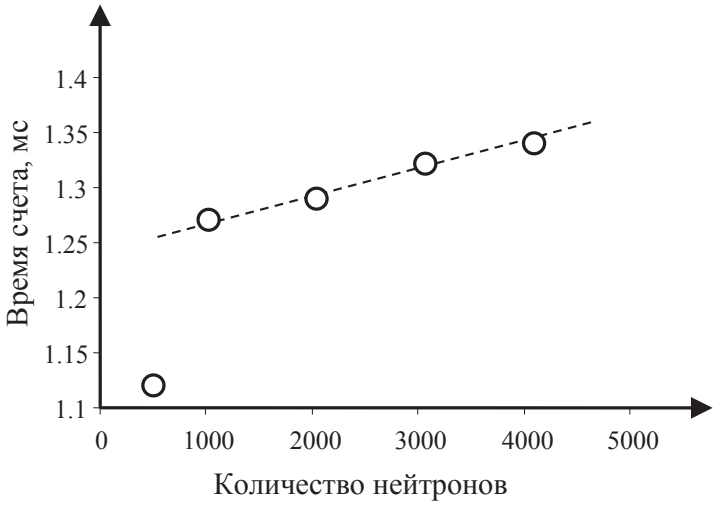



Рис. 3.4. Зависимость времени исполнения вычислительного ядра от количества нейтронов (вычислительных потоков)

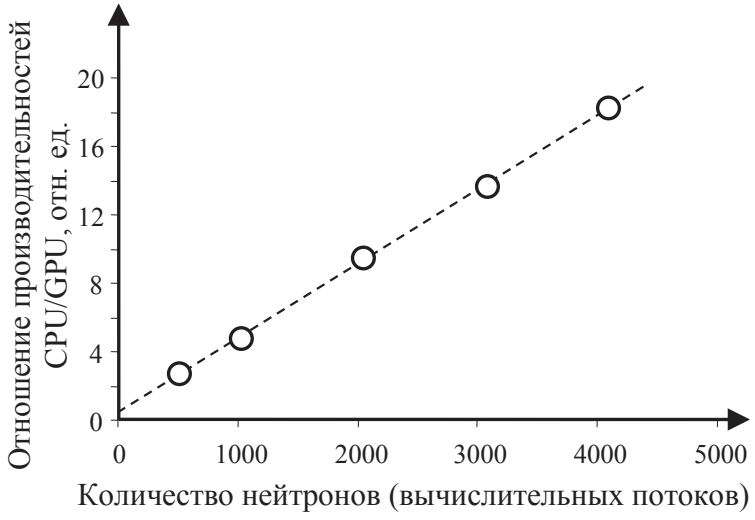



Рис. 3.5. Отношение времен исполнения вычислительного ядра на CPU и GPU в зависимости от числа параллельных вычислительных потоков

Вычислительная система, на которой были получены приводимые ниже результаты, состояла из четырехъядерного центрального процессора Intel Core 2 Quad Q9550 2.83 ГГц и графического процессора NVIDIA GeForce 8800 GTX, содержащего 16 потоковых мультипроцессоров и 128 скалярных процессоров.

Из рис. 3.4 видно, что время, затрачиваемое графическим процессором на расчет, почти не зависит от количества запускаемых вычислительных потоков: при увеличении количества потоков в 4 раза, от 1024 до 4096, время счета увеличилось только на 5.5 %. Это означает, что алгоритм из параграфа 3.2.3 не полностью задействует ресурсы графического процессора. Иначе дополнительные вычислительные потоки можно было бы запускать только за счет пропорционального увеличения времени расчета.

Рис. 3.5 показывает, что производительность расчета на GPU (то есть в параллельном режиме) хотя и выше, чем на CPU, однако не на два порядка, как для многих других приложений (см., например, [7]–[8]). Преимущество GPU возрастает с увеличением количества вычислительных потоков, но даже при 4096 вычислительных потоках достигает только 18 раз.

Для сравнения при выполнении теми же 4096 потоками генератора *Mersenne Twister* без моделирования диффузии нейтронов, GPU генерирует за единицу времени в 108 раз больше случайных чисел, чем центральный процессор. Это означает, что неоптимальный алгоритм моделирования понизил производительность графического процессора в $108/18 = 6$ раз.

### 3.2.5. Возможность оптимизации простейшего алгоритма

Если принять производительность графического процессора при исполнении программы, реализующей «чистый» генератор *Mersenne Twister* (без моделирования диффузии нейтронов), за 100 %, то получается, что алгоритм моделирования из параграфа 3.2.3 при 4096 вычислительных потоках снижает ее до 17 % ($18/108 \cdot 100\,\% = 17\,\%$). Возможно, что на самом деле ресурсы GPU задействованы еще в меньшей степени. Главной причиной является:

- то что, как только нейтрон выходит из пластины, связанный с ним вычислительный поток заканчивает свою работу. В результате

параллельные вычислительные потоки завершаются не одновременно, а так, что время работы GPU фактически совпадает со временем обработки нейтронов, находящихся внутри пластины наиболее долго. В конце расчета бóльшая часть вычислительных потоков бездействует, что и приводит к неэффективному использованию ресурсов GPU.

Для оптимизации алгоритма необходимо задействовать все запускаемые вычислительные потоки до конца расчета. В нашем примере оно может быть исполнено следующим образом;

- после выхода очередного нейтрона из пластины вычислительный поток не прекращает свою работу, а переходит к рассмотрению нового нейтрона, только что входящего в пластину, при этом:
  - координата нейтрона устанавливается равной нулю;
  - косинус отклонения траектории нейтрона μ устанавливается равным 1;
  - выполнение действий, показанных на рис. 3.3, повторяется начиная с п. 3.
- для сохранения результатов обработки всех нейтронов каждый вычислительный поток потребует трех дополнительных регистров GPU, в которые будет записывать суммарный окончательный вес отразившихся нейтронов и суммарный вес нейтронов, прошедших пластину, а также количество рассмотренных нейтронов.

Предложенная модификация алгоритма позволяет:

- задействовать все доступные параллельные вычислительные потоки до самого конца моделирования;
- синхронизировать обращения всех потоков к регистрам процессора и к видеопамяти.

Результаты моделирования диффузии нейтронов по модифицированному алгоритму (текст нового вычислительного ядра приведен в приложении) показаны на рис. 3.6. Видно, что задействование всех вычислительных потоков в течение всего времени моделирования действительно позволило резко увеличить производительность графического процессора. Теперь при 4096 вычислительных потоках преимущество GPU над центральным процессором составило 127 раз, что практически совпадает с количеством его параллельных скалярных процессоров (128).

Как показывает график (рис. 3.6), преимущество GPU над CPU существенно зависит от количества исполняемых вычислительных по-

токов: при увеличении их количества оно возрастает в несколько раз. Видно также, что преимущество в 128 раз, равное количеству скалярных процессоров в составе GPU, (по сравнению с одним CPU) не является пределом и, вероятно, даже не близко к нему (например, в нашей реализации молекулярной динамики [6] преимущество достигало 600 раз). Это означает, что графический процессор эффективно оптимизирует вычисления с использованием конвейерной обработки данных, причем для ее максимального задействования необходимо запускать на каждом мультипроцессоре больше вычислительных потоков, чем их было в рассмотренном примере (то есть больше, чем 128).

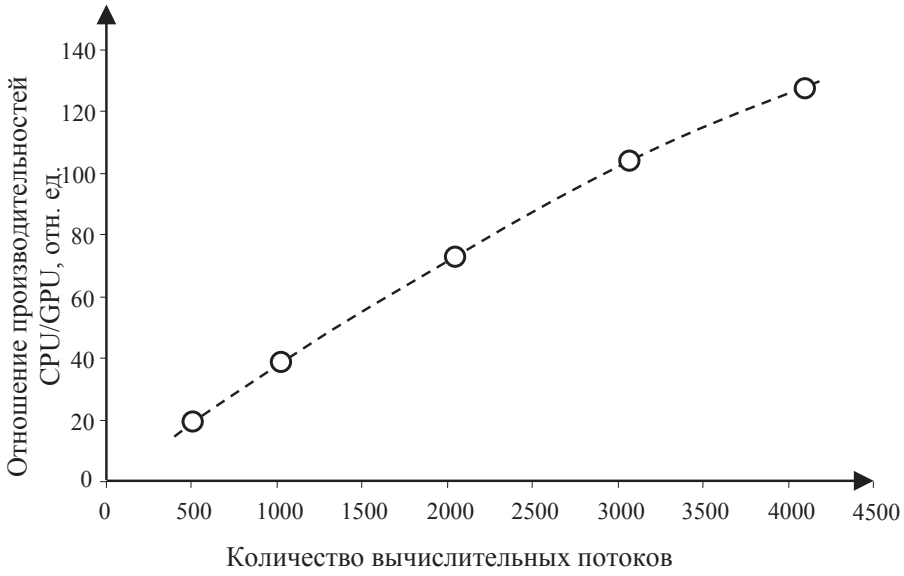



Рис. 3.6. Производительность GPU без простаивания вычислительных потоков

### 3.2.6 . Соответствие рассмотренных алгоритмов физическому содержанию задачи

При оптимизации алгоритмов физического моделирования необходимо контролировать их соответствие решаемой задаче. В частности, первый алгоритм моделирования диффузии нейтронов через пластину, который был рассмотрен в параграфах 3.2.1–3.2.3, предполагал, что все нейтроны входят в пластину одновременно, а другие нейтроны не рассматривались. Напротив, оптимизированный алгоритм из па-

раграфа 3.2.4 включает в расчет нейтроны, которые входят в пластину после выхода предыдущих.

Указанная разница в алгоритмах может давать разницу в результатах расчета, так как во втором случае на каждый нейтрон, совершающий в пластине много столкновений, приходится больше нейтронов, проходящих пластину за меньшее количество столкновений (так как нейтроны, быстро проходящие пластину, замещаются новыми). Сопоставим каждый из вариантов физическому смыслу задачи.

Предположим, что на пластину падает стационарный поток нейтронов. В таком случае внутри пластины все время будет находиться примерно одинаковое количество нейтронов (постоянная концентрация), а проницаемость и отражающая способность пластины будут определяться количеством нейтронов, проходящих и отражающихся за единицу времени. Однако в рассмотренных выше алгоритмах моделирования времяпребывание нейтрона внутри пластины вообще не отслеживается. Длительность обработки нейтрона определяется количеством его столкновений с ядрами внутри пластины. Можно принять, что в среднем длина траектории нейтрона пропорциональна количеству столкновений. Если еще скорость нейтрона остается примерно постоянной, то времяпребывание нейтрона внутри пластины (в среднем) оказывается пропорциональным количеству столкновений. В этом приближении оптимизированный алгоритм оказывается ближе к физическому содержанию задачи, поскольку заход в пластину новых нейтронов соответствует модели стационарного потока.

# Заключение

В настоящем пособии рассмотрены основные принципы поточно-параллельной реализации метода Монте-Карло на графических процессорах, совместимых с технологией CUDA. Показано, что метод Монте-Карло позволяет эффективно использовать параллельную архитектуру современных графических процессоров за счет обработки альтернативных случайных вариантов поведения моделируемой системы в большом количестве независимых вычислительных потоков. Особенности применения метода проанализированы на примере задачи о диффузии нейтронов через пластину. Продемонстрировано, что при решении этой задачи методом Монте-Карло на графических процессорах можно получить не менее чем 100-кратное преимущество в производительности по сравнению с современными центральными процессорами персональных компьютеров.

# Библиографический список


1. Metropolis, N. The Monte Carlo Method/N. Metropolis, S. Ulam // J. Amer. statistical assoc. — 1949. — V. 44. — N 247. — P. 335–341.
2. Соболь, И. М. Метод Монте-Карло / И. М. Соболь. — М. : Наука, 1985. — 78 с.
3. NVIDIA Corp. CUDA NVIDIA Compute PTX: Parallel Thread Execution [Электронный ресурс] / NVIDIA Corp. 2701 San Tomas Expressway, Santa Clara, CA 95050. — Режим доступа: http://www.nvidia.com. — Загл. с экрана.
4. CUDA C Programming Guide [Электронный ресурс]. — Режим доступа: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3k3HvOyd7. — Загл. с экрана.
5. NVIDIA CUDA Compiler Driver NVCC [Электронный ресурс]. — Режим доступа: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#axzz3k3HvOyd7. — Загл. с экрана.
6. Боярченков, А. С. Использование графических процессоров и технологии CUDA для задач молекулярной динамики / А. С. Боярченков, С. И. Поташников // Вычислительные методы и программирование. — 2009. — № 10. — С. 9–23.
7. A Survey of General-Purpose Computation on Graphics Hardware / J. D. Owens [et al.] // Computer Graphics Forum. — 2007. — V. 26. — N. 1. — P. 80–113.
8. Высокоскоростное моделирование диффузии ионов урана и кислорода в $UO_2$ / С. И. Поташников [и др.] // Сборник рефератов и докладов семинара «Вопросы создания новых методик исследований и испытаний, сличительных экспериментов, аттестации и аккредитации». — Димитровград : НИИАР, 2007. — С. 139–159.
9. Поташников, С. И. Молекулярно-динамическое восстановление межчастичных потенциалов в диоксиде урана по тепловому расширению / С. И. Поташников [и др.] // Альтернативная энергетика и экология. — 2007. — № 8. — С. 43–52.

10. Simulation of diffusion of oxygen and uranium in uranium dioxide nanocrystals / A.Ya. Kupryazhkin [et al.] // Journal of Nuclear Materials. — 2008. — V. 372. — P. 233–238.
11. Matsumoto, M. A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator / M. Matsumoto, T. Nishimura // ACM Trans. Model. Comput. Simul. — 1998. — V. 8. — P. 3–30.
12. Marsaglia, G. A current view of random numbers / G. Marsaglia // Computer Science and Statistics, Proceedings of the Sixteenth Simposium on The Interface. — 1985. — P. 3–10.
13. The pLab [Электронный ресурс] / P. Hellekalek [et al.]. — Режим доступа: http://random.mat.sbg.ac.at. — Загл. с экрана.
14. Matsumoto, M. Dynamic Creation of Pseudorandom Number Generators [Электронный ресурс] / M. Matsumoto, T. Nishimura. — Режим доступа: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf. — Загл. с экрана.
15. Podlozhnyuk, V. Parallel Mersenne Twister [Электронный ресурс] / V. Podlozhnyuk//NVIDIA Corp. 2701 San Tomas Expressway, Santa Clara, CA 95050.– 2007. — Режим доступа: http://citeseerx.ist.psu.edu. — Загл. с экрана.

# Приложение

### Оптимизированное вычислительное ядро для моделирования диффузии нейтронов методом Монте-Карло

Приведенная ниже программа представляет собой вычислительное ядро из параграфа 3.2.3, оптимизированное так, как описано в п. 3.2.4. По структуре и операциям этот код очень близок к тому, который подробно прокомментирован в параграфе 3.2.3. Поэтому здесь мы ограничиваемся только краткими комментариями, отслеживающими основные блоки программы.

```
/* Вычислительное ядро исполняется на графическом процессоре */
__global__ void MC_RandomGPU (float *d_Random, float *d_W_p, float
*d_W_m, int N_Of_Steps) {
const int tid = blockDim.x * blockIdx.x + threadIdx.x;
int iState, iState1, iStateM, iOut;
unsigned int mti, mti1, mtiM, x;
unsigned int mt [MT_NN];
/* Константы, описывающие физику взаимодействия нейтронов
с пластиной */
const float MC_h = 10.0f;//Толщина пластины
const float MC_sigma = 1.1f;//Полное макросечение взаимодействия
const float MC_sigma_S = 1.0f;/* Макросечение рассеяния, обратное средней длине свободного пробега */
const float MC_s = MC_sigma_S/MC_sigma;//Множитель веса нейтрона
mt_struct_stripped config = ds_MT [tid];//Загрузка параметров
Mersenne Twister из общей памяти
//Инициализация экземпляра Mersenne Twister
mt [0] = config.seed;
for (iState = 1; iState < MT_NN; iState++)
mt [iState] = (1812433253U * (mt [iState — 1] ^ (mt [iState — 1] >> 30))
+ iState) & MT_WMASK;
```

```
iState = 0;
/* Собственно моделирование прохождения нейтронов сквозь пластину */
float MC_x = 0.0f;//Положение нейтрона в пластине
float MC_w = 1.0f;//Вес нейтрона
float MC_m = 1.0f;//Косинус угла рассеяния нейтрона
float N_of_N = 0.0f;
float W_p = 0.0f;
float W_m = 0.0f;
for (int k = 0; k < N_Of_Steps; k++) {
//Генерация случайной длины свободного пробега нейтрона
iState1 = iState + 1; iStateM = iState + MT_MM;
if (iState1 >= MT_NN) iState1 -= MT_NN; if (iStateM >= MT_NN) iStateM -= MT_NN;
mti = mti1; mti1 = mt [iState1]; mtiM = mt [iStateM];
x= (mti & MT_UMASK) | (mti1 & MT_LMASK);
x= mtiM ^ (x >> 1) ^ ((x & 1)? config.matrix_a: 0);
mt [iState] = x; iState = iState1;
x ^= (x >> MT_SHIFT0); x ^= (x << MT_SHIFTB) & config.mask_b;
x ^= (x << MT_SHIFTC) & config.mask_c; x ^= (x >> MT_SHIFT1);
//Получение новой координаты нейтрона
MC_x -= MC_m * (__logf (((float)x + 1.0f)/4294967296.0f)/MC_sigma);
//Генерация косинуса случайной проекции нейтрона на ось x
iState1 = iState + 1; iStateM = iState + MT_MM;
if (iState1 >= MT_NN) iState1 -= MT_NN; if (iStateM >= MT_NN) iStateM -= MT_NN;
mti = mti1; mti1 = mt [iState1]; mtiM = mt [iStateM];
x= (mti & MT_UMASK) | (mti1 & MT_LMASK);
x= mtiM ^ (x >> 1) ^ ((x & 1)? config.matrix_a: 0);
mt [iState] = x; iState = iState1;
x ^= (x >> MT_SHIFT0); x ^= (x << MT_SHIFTB) & config.mask_b;
x ^= (x << MT_SHIFTC) & config.mask_c; x ^= (x >> MT_SHIFT1);
//Получение значения косинуса
MC_m = 2.0f* (((float)x + 1.0f)/4294967296.0f) — 1.0f;/* Случайный косинус угла отклонения */
if ((MC_x < 0.0f) || (MC_x > MC_h))
{
if (MC_x < 0.0f) W_m += MC_w; else W_p += MC_w;
```

```
N_of_N += 1.0f;
MC_x = 0.0f;
MC_w = 1.0f;
MC_m = 1.0f;
}
else
{
MC_w *= MC_s;//Пересчет веса нейтрона
}
}//Завершение перемещения нейтронов внутри пластины
__syncthreads ();
//Сохранение результатов расчета в общей памяти
d_Random [tid] = N_of_N;
d_W_p [tid] = W_p;
d_W_m [tid] = W_m;
}//Завершение ядра
```

# Оглавление

*Учебное издание*

**Некрасов** Кирилл Александрович
**Поташников** Святослав Игоревич
**Боярченков** Антон Сергеевич
**Купряжкин** Анатолий Яковлевич

**МЕТОД МОНТЕ-КАРЛО**
**НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ**

Редактор И. В. Меркурьева
Верстка О. П. Игнатьевой

Подписано в печать 06.05.2016. Формат 70×100/16.
Бумага писчая. Печать цифровая. Гарнитура Newton.
Уч.-изд. л. 3,0. Усл. печ. л. 4,8. Тираж 50 экз.
Заказ 124

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620049, Екатеринбург, ул. С. Ковалевской, 5
Тел.: 8 (343)375-48-25, 375-46-85, 374-19-41
E-mail: rio@urfu.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620075, Екатеринбург, ул. Тургенева, 4
Тел.: 8 (343) 350-56-64, 350-90-13
Факс: 8 (343) 358-93-06
E-mail: press-urfu@mail.ru


ДИЗАЙН УНИВЕРСИТЕТСКОГО КНИГОИЗДАНИЯ
УНИВЕРСИТЕТСКАЯ КНИГА
2015


9 785799 617233